Universitetet i Oslo
Institutt for Informatikk
D. Karabeg, I. Yu

INF2220: algorithms and data structures

Mandatory assignment 2

:

Issued: 25. 09. 2014

Due: 16. 10. 2014

1 General description: project planner

In this assignment, you are going to develop a project planning tool. A *project* consists of *tasks*, each taking an estimated *time* and certain *manpower* to complete. After task completion, the resources (manpower) are released and available to other tasks. To proceed in an speedy manner, each task in the project should start as soon as possible.

Figure 1 gives an example of a project, where a project with eight tasks is depicted as a directed graph. Each task is represented as a node with a unique identity, name, time estimate, and manpower requirements. The dependency between two tasks is represented as a directed edge.

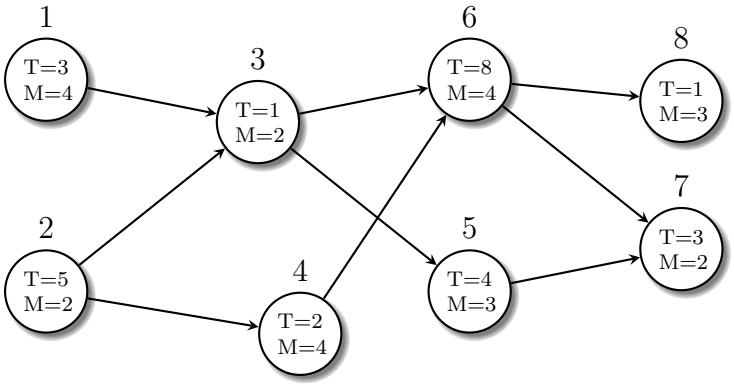

Figure 1: A sample directed graph of a project

2 Problems

The implementation should solve the following problems. Information about the input and output formats will be given later.

2.1 Realizability

Is the project realizable, considering only the task dependencies (i.e., ignoring manpower and time information). More concretely, a project is unrealizable, if the dependency graph has (at least) one cycle. E.g., the project in Figure 1 can be completed: it is acyclic (= contains no cycle). If the specified project (the input) is not realizable, your program should not just terminate with "it's impossible", but give one of those cycle as the "reason" or the symptom of the failure.

2.2 Optimal use of resources (optimizing time)

A realizable project should be done "best" = fastest possible. There will be 2 variants of the task, the second one is optional. They differ wrt. the available resources in terms of manpower.

1. *Unlimited work force:* Manpower is not a limiting factor. The time for a task is T alone.

2. *Limited work force (optional):* Only a specified, limited number of persons are available, which limit the tasks that can be executed at the same time.

You are asked to come up with a "work-plan" or schedule which "optimizes" the usage of time.

Optimize overall completion time & unlimited manpower Tasks can be scheduled to be done "in parallel", e.g., tasks 1 and 2 can (and should therefore) start as soon as the project starts. The rest of the tasks will only start until all of its predecessors are finished. For example, task 3 has to wait until task 1 and 2 are both completed before it starts. It is assumed that there is never a shortage of manpower, so whether a task can start only depends on the completion of the tasks it depends upon.

Sometimes, tasks can be *delayed* without delaying the *overall* project completion time because other tasks need more time anyhow. For instance, in Figure 1, task 1 may wait up to 2 units without delaying the project because task 3 still has to wait for 2. Note further that *any* delay of task 3 in turn will delay the overall completion. Tasks which can not be delayed, such as task 3 in the picture, are called *critical*. Note also that being critical is not equivalent to the fact that postponing the node causes a delay of the tasks *immediately* following; as said, a task is critical means a delay of the node delays *the whole project.*

1. How can the project be accomplished in the shortest possible amount of time?

2. Give also the corresponding optimal time.

3. How much manpower will be used at any given time?

4. Which tasks are *critical.*

Optional: limited manpower Tackle the problem under the restriction of limited manpower. A task can start *only* when all tasks it depends on have completed (as before) and if enough people are available, i.e., not busy with other tasks. The question of "criticality" is not part of this optional assignment.

It is actually (computationally) hard to calculate an optimal schedule. The task therefore is not even to attempt that, but make a "common sense" approach (one could say, a *heuristic* approach) which generalizes the one for unlimited manpowers: simply start at each point as many tasks as possible and this way at least avoid "idling time" where tasks *could* be started but are not yet handled.

Remark 1 (Limited manpower) As a remark concerning the "algorithmic design principle": one might say, the sketched strategy, both for limited as for unlimited manpower, is a *greedy one:* to optimize the overall project duration, simply choose at each point "greedily" what currently appears to be the most promising course of action, namely: start right away. If there is limited manpower, the strategy to "start right away" is still a good one; however, due to restricted manpower, there are situation where not all tasks which are in principle ready to be tackle right away as far as their dependencies are concerned, cannot for lack of manpower. Therefore, the algorithm has *to choose* among those. Unfortunately a choice may turn out to be non-optimal later (one "regrets" the choice) and to find out the optimal solution, one may explore all alternatives. That makes this version of the problem harder, if one aims for the optimum. But as said: when you choose to tackle that generalization, don't aim for a global optimum, just schedule as much tasks possible and as soon as possible.

3 Guidelines & hints for the implementation

Reading input files and constructing the graph

A project is given by an input file. At the top of this file is the total number of tasks comprising the project. The data format of the input file will be a list of task definitions, where each of them is a sequence of the following data:

Identity of this task	integer
Name of this task	a string
Time estimate for this task	integer
Manpower requirements	integer
dependency edges	A sequence of identities that states which tasks must be completed before this task can start (its dependencies). This list is terminated by a 0.

To represent the graph we can use the following class:

```
class Task {
    int      id, time, staff;
    String   name;
    int      earliestStart, latestStart;
    Edge     outEdges;
    int      cntPredecessors;


}
```

Note that this is just a skeleton of the classes. Additional fields may be needed according to your implementation. Each task should be an object of the class "`Task`". After the input file has been read, it should contain the right values for: id, name, time, staff and cntPredecessors. You can take it for granted that all task numbers lie in the interval `1` to `maxnr` where `maxnr`, which is the total number of tasks comprising a project, is given at the top of the input file.

For each task, there should be a list of `Edge`-objects which represents the outgoing edges from this task (i.e., the tasks that cannot start up before this task is finished). As the input file is read, these data structures are constructed along the way. One has to remember to instantiate a new `Task`-object the first time a task is mentioned in the input file, which can very well be in a predecessor-list, before the definition of the task. Just leave the data-fields of this `task`-object blank until its definition occurs.

3.1 Handling circular dependencies

Realiziability from Section 2.1 requires to check for cycles in the graph. If a circular dependency is detected, the program should exit, tell the user that this project can not be completed, and print the detected circular dependency. This should be checked as soon as the graph is constructed.

If no circular dependency is detected, we should proceed to give optimal schedules.

3.2 Parallel tasks: Start tasks as soon as possible

One should start up every task as soon as possible, that is, after all tasks it depends upon are completed. Tasks without dependency should be started right away. Output should be provided again by printing out important information, i.e., when tasks start up and/or finish. Your system should also print out current working staff at these moments in time. For the project illustrated in figure 1, feedback from the system should be something as follows:

Listing 1: Suggested output

```
Time: 0     Starting: 1
            Starting: 2
       Current staff: 6

Time: 3     Finished: 1
       Current staff: 2

Time: 5     Finished: 2
            Starting: 3
```

```
            Starting: 4
       Current staff: 6

Time: 6     Finished: 3
            Starting: 5
       Current staff: 7

Time: 7     Finished: 4
            Starting: 6
       Current staff: 7

Time: 10    Finished: 5
       Current staff: 4

Time: 15    Finished: 6
            Starting: 7
            Starting: 8
       Current staff: 5

Time: 16    Finished: 8
       Current staff: 2

Time: 18    Finished: 7

**** Shortest possible project execution is 18 ****
```

3.3 Finding critical tasks

To find out whether a task is critical to complete on time, check if there is any "*slack*" for the task. To achieve that, we have to know when a task can be started at the *earliest*, and when it has to be started at the *latest*. The difference between these two time points is the *slack* of a task. A task that does not have any room for delay, i.e., a task that does not have slack, is considered to be *critical*. Your program should show the user a list of all the tasks (sorted by their identifying number) which includes these properties:

- Identity number
- Name
- Time needed to finish the task
- Manpower required to complete the task
- Slack
- Latest starting time
- A list of tasks (identities) which depend on this task

4 How to deliver

The assignment should be carried out individually and delivered through Devilry `https://devilry.ifi.uio.no/`.

- the implementation language is JAVA.
- Your implementation should compile on the LINUX machines in the University, and run either with the command:

  ```
  java Oblig2 <projectName>.txt manpower
  ```

  where first argument of the `Oblig2` program is input file specifying the project, and the second argument `manpower` is an integer.
- Your program is not expected to interact with the user while running; it just takes its input from the command line (file name and manpower argument), and print out the answers to the assignment questions on the command window.
- Note that the value of the parameter `manpower` would be `999` to represent unlimited manpower. If you want to try the extra question, you can use `8` as the value of `manpower` for project `buildhouse1` and `buildhouse2`, and `100` for project `buildrail`. The corresponding input file of the mentioned projects can be found from the course webpage.
- The delivery must only be in one file: either a `.tgz` or `.zip` archive (with one of those two file extensions). Follow the steps below to generate the archive:

  1. `cp AssignmentFolder myusername`
  2. `find myusername -name *.class -delete`
  3. `tar cvzf myusername.tgz myusername`

  That is, first you copy the contents of `AssignmentFolder` to a folder with the name `myusername`, then delete all compiled files from the `myusername` folder and pack the folder in a gzipped tar. Of course, you have to create the directory `myusername` before step 1 from above. For a `zip`-archive, replace the `tar`-command in step 3 with: `zip -r myusername.zip myusername`

Your archive should contain:

- The source file(s) of your implementation.
- A **report** in which you should state the complexity of your implementation and justify the stated complexity. If you have different complexity for each question, discuss them separately.
- A file named `output.txt` in which you put the feedback (print out) from system during execution of project `buildhouse1`, `buildhouse2` and `buildrail`.
- A file named `README` in which you should describe any peculiarities of your solution, for instance, things that may be missing, or assumptions made, or questions for the group teacher. If everything runs fine, put it down in the `README` file.

Good luck!